WS-TAXI: a WSDL-based testing tool for Web Services*

Cesare Bartolini, Antonia Bertolino, Eda Marchetti
Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo"
Consiglio Nazionale delle Ricerche - Via Moruzzi 1 - 56124 Pisa, Italy
{cesare.bartolini, antonia.bertolino, eda.marchetti}@isti.cnr.it

Andrea Polini
Dipartimento di Matematica ed Informatica, University of Camerino
Via Madonna delle Carceri, 9 - 62032 Camerino, Italy
andrea.polini@unicam.it

Abstract

Web Services (WSs) are the W3C-endorsed realization of the Service-Oriented Architecture (SOA). Since they are supposed to be implementation-neutral, WSs are typically tested black-box at their interface. Such an interface is generally specified in an XML-based notation called the WS Description Language (WSDL). Conceptually, these WSDL documents are eligible for fully automated WS test generation using syntax-based testing approaches. Towards such goal, we introduce the WS-TAXI framework, in which we combine the coverage of WS operations with data-driven test generation. In this paper we present an early-stage implementation of WS-TAXI, obtained by the integration of two existing softwares: soapUI, a popular tool for WS testing, and TAXI, an application we have previously developed for the automated derivation of XML instances from a XML schema. WS-TAXI delivers a complete suite of test messages ready for execution. Test generation is driven by basic coverage criteria and by the application of some heuristics. The application of WS-TAXI to a real case study gave encouraging results.

1 Introduction

Service-oriented Architecture (SOA) is the emerging paradigm to enable interoperability and flexibility of distributed applications. All major IT vendors, such as IBM, Tibco, Software AG, Oracle, have made huge investments into SOA in the last years, making up a global estimated budget of $2 billion in 2007, which is further expected to rise and reach $9.1 billion by 2014 [11]. Beyond such numbers is the fact that enterprises in virtually any domain, banks, governments, hospitals, academies, leisure and travel agencies, are progressively shifting towards the on-line service-market.

Our research addresses the testing of Web Services (certainly the most widely adopted technology to implement a SOA), which has been recognized as a topmost critical issue for the IT industry of the future: for instance, a recent study by Gartner lists insufficient validation in the 'hit list' of the most common technological errors in planning SOA implementations [10]. The same study recommends that at least 25% of the effort spent in a SOA project is dedicated to testing.

Indeed, due to their pervasive distribution, services must offer strict guarantees of reliability and security. Therefore, WSs need to be thoroughly tested before deployment, and several industrial testing tools specialized to WSs technology are today available, such as soapUI [9], PushToTest [15], and SOATest [14], just to mention a few.

Essentially, a WS collects a set of functions, whose invocation syntax is defined in an associated WSDL (WS Description Language) [22] document. The formalized WSDL description of service operations and of their input and output parameters can be taken as a reference for black box testing at the service interface.

As an example, soapUI, which is acclaimed on its distribution site [9] as the most used tool for WSs testing, can automatically produce a skeleton of a WS test case and provide support for its execution and result analysis. The tester's job is certainly greatly released

*The authors wish to thank Antonino Sabetta for his contribution in defining the test cases. This work was supported by the EU FP7 Project 216287 TAS^3 and by the Italian MIUR PRIN 2007 Project D-ASAP.

by the usage of this or similar tools; however the produced test cases are incomplete and lack the input parameter values and the expected outputs. Moreover, soapUI can also measure the coverage of WS operations, but again the generation of diverse test messages for adequately exercising operations and data combinations is left to the human testers.

It is somewhat surprising that till today WS test automation is not pushed further than this, since in principle the XML-based syntax of WSDL documents could support fully automated WS test generation by means of traditional syntax-based testing approaches. In this direction, we hereby propose a framework for "turn-key" generation of WS test suites. Our approach easily realizes a practical yet powerful tool for fully automated generation of WSs test inputs. The key idea, sketched in [5], is to combine the coverage of WS operations (as provided by soapUI) with well-established strategies for data-driven test input generation.

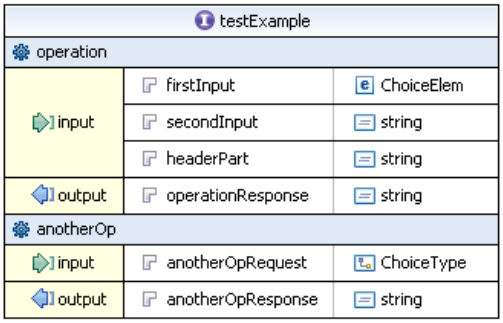
In this paper we present the logical architecture and an early-stage implementation of the proposed framework. The prototype is obtained by integrating two existing softwares: soapUI, already presented above, and TAXI [20], which is an application we have previously developed for the automated derivation of XML instances from an XML schema (the interested reader can refer to [20] for details on TAXI implementation) The new integrated framework is named WS-TAXI. The original notion at the basis of WS-TAXI, in comparison with soapUI and other existing WS test tools, is the inclusion of a systematic strategy for test generation based on basic well-established principles of the testing discipline, such as equivalence partitioning, boundary analysis and combinatorial testing.

The paper is structured as follows. In the next section we explain the motivations of the approach; in Sec. 3 we present the WS-TAXI methodology and its current implementation; in Sec. 4 we then demonstrate the functioning and effectiveness of WS-TAXI on a case study. Related work is surveyed in Sec. 5 and conclusions are drawn in Sec. 6.

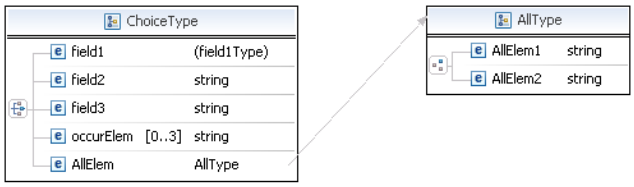
2 Motivation

Before describing our approach, in this section we provide a motivating example. We first show the type of support provided by soapUI [9], and then explain how our proposed approach improves on it.

Let us take a simple example of a WSDL specification for a dummy *testExample* service. The WSDL, as shown in Figure 1a, specifies two operations, called *operation* and *anotherOp*. The former accepts an input message composed of three parts:



(a) WSDL port type



(b) XSD excerpt. ChoiceType is also the type of ChoiceElement

Figure 1. Simple WSDL example.

- *firstInput* which is an element, called *ChoiceElement*, of a user-defined complex type *ChoiceType*. In particular, it is a *choice* element with five possible choices, one of which is an *all* element called *AllElem*, as shown in Figure 1b;
- *secondInput*, a basic XSD type (string);
- *headerPart*, another XSD string type which is used in the SOAP header.

The second operation accepts messages with a single part (*anotherOpRequest*) which is of the user-defined complex type *ChoiceType* (Fig. 1b). Our WSDL specification also defines the response messages of the two operations, which are *operationResponse* and *anotherOpResponse*, respectively.

If we run the soapUI tool on this WSDL file, it automatically generates two separate SOAP envelopes, one for each operation, containing the skeletons of some test input. Figure 2 shows the one for *operation*. As we can see, soapUI leaves question marks (or alternatively some pregenerated Latin words or numbers) in the place of actual input data for the generated test cases (see lines 13–15, 17, 21–22, 25 in Fig. 2); in addition, it does not manage the *choice* and *all* elements and the occurrence attributes, but introduces comments (see lines 11–12, 19–20 and 16 in Fig. 2, respectively) explaining to the human tester how to handle them. Finally, the tool distinguishes the basic data types but with little variability in the kind of

```
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/"
   xmlns:test="http://www.example.org/test/">
  <soapenv:Header>
      <headerPart>?</headerPart>
   </soapenv:Header>
   <soapenv:Body>
      <test:operation>
         <test:ChoiceElem>
            <!--You have a CHOICE of the next 5 items
                at this level-->
            <field1>?</field1>
            <field2>?</field2>
            <field3>?</field3>
            <!--0 to 3 repetitions:-->
            <occurElem>?</occurElem>
            <AllElem>
               <!--You may enter the following 2
                   items in any order-->
               <AllElem1>?</AllElem1>
               <AllElem2>?</AllElem2>
            </AllElem>
         </test:ChoiceElem>
         <secondInput>?</secondInput>
      </test:operation>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 2. Skeleton generated by soapUI

value used for instances derivation. For instance, soapUI fills date or numeric types using the same value for all instances, while it uses predetermined combinations of Latin words for string data. For completeness' sake, we add that the commercial version of the tool improves this aspect a bit and provides some support for data generation. However, as far as we could experiment with it, this function only deals with simple types and still requires complex user guidance to properly manage the input selection.

It is evident from this simple example that the tester still needs to do most of the work before such SOAP skeletons can be effectively launched as test cases. Our contribution starts where soapUI stops: WS-TAXI can deliver ready-to-launch test messages by automatically instantiating XML instances for the XSD structures included in the WSDL specification (and therefore getting rid of all the question marks or bogus words in the soapUI test skeleton).

Some hints of how the example in Figure 1 is processed by WS-TAXI are:

- the *choice* within the *firstInput* element is substituted by systematically picking every possible child element;
- *AllElem* is substituted with random orderings of its child elements;
- for *occurElem*, which specifies 0 as *minOccurs* and 3 as *maxOccurs*, three different results are instantiated: we take the minimum, the maximum and an intermediate number of occurrences (when unlimited, a maximum bound is set by the user before instance generation);
- the string types are dealt with by generating compliant input data.

```
<?xml version="1.0"?>
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/"
   xmlns:test="http://www.example.org/test/">
  <soapenv:Header>
    <headerPart>ydykDkjj</headerPart>
  </soapenv:Header>
  <soapenv:Body>
    <test:operation>
      <ChoiceElem>
        <occurElem>gTqNyYyb</occurElem>
        <occurElem>xEXWdpVs</occurElem>
        <occurElem>o</occurElem>
      </ChoiceElem>
      <secondInput>YjuyjuLd</secondInput>
    </test:operation>
  </soapenv:Body>
</soapenv:Envelope>
```

```
<?xml version="1.0"?>
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/"
   xmlns:test="http://www.example.org/test/">
  <soapenv:Header>
    <headerPart>ydykDkjj</headerPart>
  </soapenv:Header>
  <soapenv:Body>
    <test:operation>
      <ChoiceElem>
        <field1>value2</field1>
      </ChoiceElem>
      <secondInput>YjuyjuLd</secondInput>
    </test:operation>
  </soapenv:Body>
</soapenv:Envelope>
```

```
<?xml version="1.0"?>
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/"
   xmlns:test="http://www.example.org/test/">
  <soapenv:Header/>
  <soapenv:Body>
    <test:anotherOp>
      <anotherOpRequest>
        <AllElem>
          <AllElem2>UwiiuPrA</AllElem2>
          <AllElem1>IhESSchU</AllElem1>
        </AllElem>
      </anotherOpRequest>
    </test:anotherOp>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 3. Messages generated by WS-TAXI

Applying such rules, for every part of every input message WS-TAXI can generate a number of instances (at least as many as all the possible structures of the part itself or its types). Input messages are then created by suitably selecting instances of the various parts

composing the message, both in the SOAP header and body. Ideally, this would require a cartesian product of the instances of all parts; since this can quickly get out of control, in generating the messages we apply some heuristics inherited from the combinatorial and pair wise testing [1] to draw without repetitions all different instances, until all of them have been used. Finally, the SOAP envelopes are populated with the generated message instances, as above.

The application of WS-TAXI to the example of Figure 1 would generate seven structurally different SOAP envelopes, which may be filled by varying the input values, originating an arbitrary number of instances. A sample of three of them is shown in Fig. 3. In particular, the first sample is an invocation to *operation* where the *occurElem* has been chosen, with three occurrences. The second is another invocation to *operation* where *field1* was chosen, and a specific value was randomly selected from an enumeration. The third sample is an invocation of *anotherOp* where the *AllElem* has been chosen, and a random ordering has been assigned to the elements in the *AllType*. Concerning the string type, unless the type is constrained by an enumeration in the schema, the possible options are either a random generation (as is the case for Fig. 3), or more meaningful values retrieved from a database previously populated for that purpose.

By comparing our generated test cases (Fig. 3) with the results produced by soapUI (Fig. 2), the gain in automation is evident. While soapUI generates sketchy outlines or very simple SOAP envelopes, our approach delivers a more complete test suite which spans over the several possible structures compliant with the associated XML Schema.

Increased automation comes coupled with flexibility: in WS-TAXI the tester is provided with several handles through which the test generation process can be tuned. The test suite generation can be driven by basic coverage criteria and by the application of some test heuristics. Moreover, by systematically combining the generated element instances in different ways, and by varying the number and data values of the instances, the tester can automatically obtain as many test messages as desired. The approach is detailed in the next section.

3 Approach

In this section we detail the WS-TAXI framework. The main activities performed by WS-TAXI are:

WSDL Analysis The WSDL specification is parsed and useful information, such as operations, messages and the data structures (XSD), is automatically extracted[1].

SOAP Envelope Derivation For each operation a separate SOAP envelope is automatically extracted. It does not contain ready-to-send messages so far, but only their skeletons.

Definition of Message Parts For each data structure in the WSDL specification, different message instances are generated.

Composition of Envelopes The bogus data in the envelope skeletons are replaced with the actual derived instances.

Message Sending and Results Analysis The built envelopes are sent to the service. The output messages are collected and submitted to inspection.

We have realized a working prototype which incorporates soapUI and TAXI. The logical architecture of the WS-TAXI framework is depicted in Figure 4. The WS-TAXI activity starts taking the WSDL specification of the service to be tested as an input. This is given both to the soapUI and `getXSD` components. The former is responsible of the SOAP envelope skeleton derivation, while the latter extracts the schemas referenced by or contained within the WSDL and passes them to the TAXI component. This last one is in charge of the actual message definition and will be described in detail in Section 3.3. Once available, the XML instances derived by TAXI and the envelope skeletons generated by soapUI are both given to the `Composer` component for assembly. Different testing coverage criteria can be adopted by `Composer`, as described below. Besides, since this methodology can lead to an exponential explosion of the number of generated envelopes, some heuristics to forcibly control the number of test cases are also implemented. Finally the `Message Sender` component sends the envelopes one by one to the server and collects the results. As said, the test oracle is not (yet) automated: the test results are presented to the tester or could be checked against provided expected output annotations (as is done also by soapUI).

3.1 Coverage criteria

As mentioned, WS-TAXI can be used for testing a service under different levels of thoroughness, which correspond to the different coverage criteria adopted by the `Composer` component:

Operation Coverage: For each envelope skeleton provided by the soapUI component, corresponding to a

[1] We conducted our experiments considering WSDL using *RPC style*. Nevetheless the tool should be applicable without major revision to *Document style*.

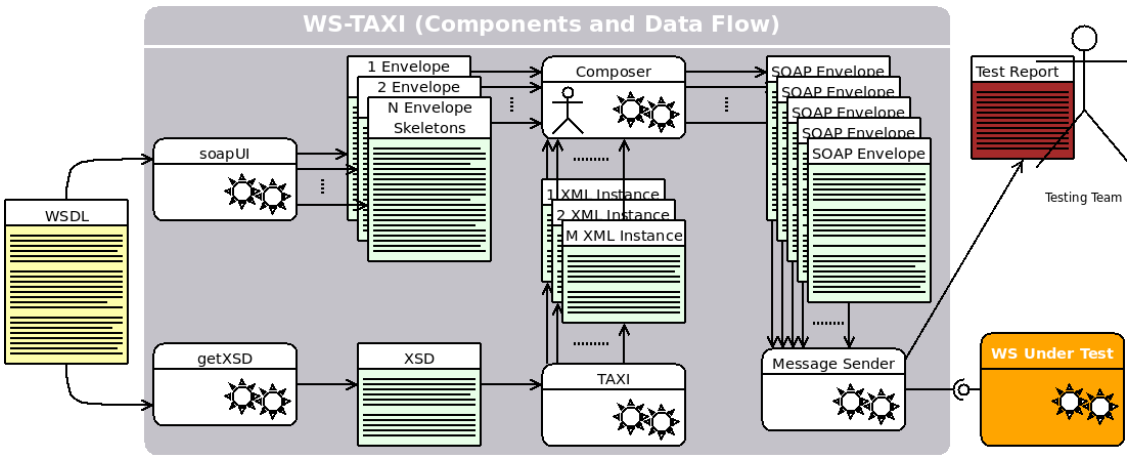


Figure 4. Diagram of the WS-TAXI approach.

different operation of the WSDL specification, a single TAXI-generated message is packaged into the envelope (it is picked using a pseudo-random algorithm). For instance, with respect to the WSDL depicted in Figure 1, two messages are required to achieve operation coverage, one for *operation* and one for *anotherOp*. The first (or second) and third samples in Figure 3 would suffice.

Message Coverage: Among the messages declared in the WSDL specification only those used as inputs of the operations are taken into account for testing purposes. For this reason it is necessary to select a representative (sub)set of operations so that all the different input messages are exploited at least once. As an example, consider a set of operations Op1, Op2 and Op3 so that Op1 uses messages A and B, Op2 uses messages C and D, Op3 uses messages A and C; in this case, Op1 and Op2 are sufficient to cover all the possible input messages (A, B, C, D). Then, for each message a suitable instance is defined and packaged into the proper operation envelope. Message instances are selected from those provided by the TAXI component. The coverage of input messages implicitly determines the coverage of their parts. Considering Figure 1, since the two operations use different input messages, message coverage does not differ from operation coverage.

Schema Coverage: The parts composing each message could be associated to an XSD type or element. For each data specification, a set of instances is derived. The instances may vary either in structure or in value. Note that to cover the full set of the XSD structures, a subset of the messages required as input operations might be sufficient. From this observation, we conceived the mixed approach below. Using the sample WSDL from Figure 1, since the input message for *operation* contains a *choice* with an element with *minOccurs* and *manOccurs* attributes, seven instances are needed to cover that schema (one for each child of the *choice*, except for the *occurElem* which needs three instances, as described in Section 2). In particular, the first sample in Figure 3 shows the *occurElem* selection with the maximum allowed occurrences, while the second one shows the *field1* selection; we omit the remaining ones due to space constraints. Additionally, the same set of instances is sufficient to cover the schema for the input message of *anotherOp* also (because *anotherOp* uses only the *ChoiceType*), so there is no need to have instances related to that operation.

Mixed Approach: The two criteria of operation coverage and schema coverage are combined together for deeper analysis. Specifically, each operation is selected at least once, and in addition, where applicable, each operation is invoked repeatedly so as to cover all the possible data structures of the input messages. Again, following our example, to achieve the mixed approach it is necessary to make a schema coverage of all messages. This implies using the seven instances from the schema coverage, plus seven more for covering the message of *anotherOp*. One of the additional instances is the third sample in Figure 3.

Concerning the relationships among the proposed criteria: Operation coverage implies message coverage but not the opposite. Operation and message coverage guarantee that each schema type or element has been exercised at least once but do not guarantee that every possible structure has been generated for each type or element. Schema coverage does not automatically imply message or operation coverage.

In the rest of this section we further detail the role of the soapUI and TAXI tools into the WS-TAXI framework.

3.2 soapUI

soapUI [9] is a tool developed by Eviware Software AB, available both in free and improved commercial versions. It assists programmers in developing SOAP-based web services. In particular, it can:

- generate stubs of SOAP calls for the operations declared in a WSDL file;
- send SOAP messages to the web service and display the outputs;
- mock a web service, by providing a listening server compliant with a WSDL file;
- populate a data source and generate messages with data extracted from it (only in the commercial version; this feature requires a certain amount of domain knowledge and skill with the tool usage. Furthermore *choices*, *all*, *occurrences* and other constructs are not automatically dealt with);
- run batch tests, and (in the commercial version) produce some partial information about coverage of the data value sets used in the operations.

For the purposes of this research we mainly used the first feature, and, to a minor extent, the second one for some preliminary tests.

3.3 TAXI

TAXI (Testing by Automatically generated XML Instances) [6, 20] is a tool able to generate compliant XML instances from a given XML Schema. It has been conceived so as to cover all interesting combinations of the schema by adopting a systematic black-box criterion. For this reason, TAXI applies the well-known Category Partition (CP) technique [13] to the XML Schema. CP provides a stepwise intuitive approach to identify the relevant input parameters and environment conditions and combine their significant values into an effective test suite.

Here below we present the main activities of the TAXI component. TAXI activity starts with the analysis of an input XML Schema. In case *choice* elements are included into the schema, a set of sub-schemas are derived by selecting a different child from each *choice* element. In the likely case that more than one *choice* elements are present, a combinatorial methodology of *choice* children is performed. This ensures that the set of sub-schemas represents all the possible structures derivable from *choice*.

The implementation of CP requires the analysis of the XML Schema and the extraction of the useful information. Element occurrences and types are analyzed, and the constraints are determined, from the XML Schema definition. In particular, boundary values for *minOccurs* and *maxOccurs* are defined: for instance if *maxOccurs* is associated to "unbounded", a suitable value is used; if specific values are defined for *minOccurs* and *maxOccurs*, they are used as boundary values. Exploiting the information collected so far and the structure of the (sub)schema, TAXI derives a set of intermediate instances by combining the occurrence values assigned to each element.

The final instances are derived from the intermediate ones by assigning values to the various elements. Two approaches can be adopted: values can be picked from a database embedded within TAXI or generated randomly if no value is associated to an element in the database. In the former case the database should be previously populated with meaningful values for each element. The database does not play an active role in the testing process; it only serves as a data source for building the instances. Since the number of instances with different structures could be huge, in the current implementation TAXI only selects one value per element for each instance.

During the final instance derivation, particular care is devoted to the *all* elements. Each time an *all* construct is in an intermediate instance a random sequence of the *all* children elements is chosen for generating the final instance. This new sequence is then used during the assignment of values to each element.

In order to make the generation more flexible, TAXI also provides different test strategies to pilot the instance generation. This requires, for instance, either the coverage of the possible (sub)schemas, or occurrence combinations, or value combinations, or simply the generation of a prefixed number of instances. For space constraints we do not provide further details but we refer to [6] for a more accurate description.

3.4 What WS-TAXI does not do

In the paper, we have sometimes used the term "test suite" to refer to the set of test messages generated by WS-TAXI, but we did so somewhat improperly. Since the generation is only based on the WSDL syntax, in its current version WS-TAXI only produces ready-to-launch test input messages, such as those shown in Fig. 3, but cannot also automatically produce the test oracle. As said, presently the test results are collected and presented to the tester or checked against user-provided annotations. However, all the generated mes-

sages are compliant by construction to the WSDL interface. Hence, even though we cannot predict the service response, we know at least a partial oracle, that is, it is mandatory that all WS-TAXI test messages be accepted by the service under test.

Another natural limitation of WS-TAXI, as well as of any other syntax-based test approach, is that the generated test messages do not consider possible dependencies between messages, i.e., WS-TAXI is (as yet) agnostic of any protocol. For such necessities, other test approaches, based on behavioural specification, should be considered.

4 Case Study

To measure the strength of the proposed approach, our methodology has been applied to a web service which queries a publications database. The service is described in Sec. 4.1, and the empirical study in Sec. 4.2.

4.1 The Pico service

Pico is a web-based PHP application for scientific publication management currently in use at Scuola Superiore Sant'Anna [2]. Registered users can add, edit and remove the publications in the database, however its information is publicly available, therefore search results can be viewed by non-registered users. Beyond direct connection to the database, searches can be executed through the provided web service using SOAP invocations. Tests were conducted on a system running a development version of the Pico software on a subset of the real publications database. The service, based on a SOAP RPC binding, runs on PHP 5.2.5 using the native SOAP module, and the underlying database application is MySQL 5.0.45 Community Edition. For the sake of simplicity, although the service is composed of a high number of operations for different purposes, only three of them are used in this experiment (for a total of 215 PHP LOCs) and included in the WSDL file processed by WS-TAXI:

searchByAuthor performs a search of all the publications in the Pico database including a given name in their list of authors.

- Inputs: author's name (type: string); research sector. If not specified, then all research sectors will be included in the search (type: string); year (type: string); publication type (journal, conference proceedings...). If not specified, then all publications from this author will be returned (type: string).
- Outputs: a list of references. If no publications relative to the requested author were found an empty element is returned (type: sequence of record elements, from a custom XML Schema Definition).

searchByTitle performs the search according to the requested title.

- Inputs: publication title, or part thereof (type: string).
- Outputs: a list of references identical to the previous function.

pdfGetter recovers the file containing the printable form of the publication and returns its URL. It is important to note that this function does not actually return a PDF file, but a publicly accessible URL, so the requestor has a greater flexibility in choosing how to recover the actual file.

- Inputs: the name of the file (type: string).
- Outputs: the full URL where the file can be retrieved, or an empty value if it is not available (type: string).

4.2 Experimentation

To evaluate the benefits provided by a new test methodology, it is usual to generate mutants of a system under test and measure how many of them are killed. In our case study, a "trusted" version of the Pico web service has been used as a basis, and a number of mutants has been generated from it. We have not found any mature or consolidated tool for PHP mutations, therefore we have generated a number of mutants by ourselves, using the following standard rules for mutation:

- *statement deletion*: Every statement that could be deleted without leading to incorrect PHP code has been commented out, generating a separate mutant, for a total of 105 mutants;
- *booleans*: every `if` statement has been forced to `true` and `false`, generating two mutants for each such statement, for a total of 32 mutants;
- *logical operators*: every $<$, $<=$, $>$ and $>=$ operator has been replaced with the other three and with $==$, giving 4 mutants each, for a total of 8 mutants;
- *boolean operators*: every $==$ or $!=$ operator has been replaced with its opposite, giving a total of 8 mutants.

The total number of mutants is 153. Another usual problem in evaluating a new test methodology is to compare it against a relevant baseline method, to show how the new approach improves over the old one. In our case, we are not aware of any other approach that can automatically derive a systematic test suite as we do. We therefore decided to compare our automated test suite against a manually generated one, mimicking a tester using the soapUI tool.

Against the generated mutants, two parallel testing approaches have thus been undertaken. On one side, for each operation, 4 custom test cases were built ad-hoc from soapUI skeletons by an expert tester. On the other side, for a fair comparison, the same number of SOAP calls were generated using WS-TAXI, under the mixed coverage approach (see Section 3). Both test suites were made up exclusively of XSD-compliant messages, and included both data actually taken from the publications database, and fictitious names or keywords. A total of 12 operations for each test suite were therefore run.

As a result, we observed that the custom test suite managed to kill 65 mutants, while the WS-TAXI test suite killed 106. A deeper analysis revealed a generalized result for the custom test suite, meaning that the non-killed mutants belonged to all of the four previously defined categories; on the other hand, the remaining mutants for the WS-TAXI test suite were concentrated among those created with the *booleans* rule.

In the above test, however, WS-TAXI was underutilized: employing this tool for such a low number of tests is a nonsense. We then progressively increased the number of SOAP calls generated by WS-TAXI, and run an extended version of the previous test. We did not do the same for the custom tests, due to the excessive effort required to manually build a lot of SOAP envelopes.

With 20 instances for each of the three operations, we did not obtain any improvement. With 50 instances per operation, we killed an extra 2 mutants. With 100 and 200 calls per operation, we managed to kill 3 additional mutants, for a total of 111. A deep static analysis of the code of the remaining 42 mutants led to the conclusion that further increasing the number of tests would not manage to kill any more mutants. A summary of the results is shown in Table 1.

The 42 non-killed mutants were thus classified:

- 22 mutants were actually equivalent. PHP does not enforce declaring arrays, for example, therefore deleting such a statement causes a performance loss, but does not compromise the results. Since PHP is a loosely-typed language, there were several mutants which would slow down performance or issue some warning (if warnings were enabled) but are actually equivalent from the point of view of functionality;
- 18 mutants were also equivalent, but they helped evidence two small bugs in the source code of the original application. This will be explained in greater detail later in this section;
- there were 2 mutants which our test suite was not able to distinguish, so they remained alive.

Tests	Killed	Not killed	Percentage
12 (manual)	65	88	42.48%
12	106	47	69.28%
20	106	47	69.28%
50	108	45	70.59%
100	111	42	72.55%
200	111	42	72.55%

Table 1. Summary of the WS-TAXI results.

The first result that needs some thoughtful consideration is the fact that the automatically-generated test suite is actually more efficient than the manual one. Reasonably, given enough time, a professional tester could achieve results much similar if not equivalent to those of the automated test suite. However, manually writing test cases takes much longer than an automated exhaustive XML-based generation, and might not be sufficient to cover the whole spectrum of criticalities. An interesting side aspect is that running the automated test suite with 200 instances per operation still required much less time than the 12 instances per operation manually written by the tester.

A thoughtful analysis of the results proved that the 2 alive mutants were missed not directly because of issues in the WS-TAXI methodology, but rather because of the sample data with which the TAXI database had been populated. There were no data values covering the specific differences of those mutants from the original: indeed, this further confirmed us the importance of producing a good number of input messages varying the data values, as can be done with WS-TAXI.

An important side effect of this analysis was the opportunity to detect two bugs in Pico which had not popped out before. One of these errors was related to some search parameters which were not passed to the search function; this parameters would act as a filter on the search results, and their absence returns some results which may not be requested in the search. The second error was the lack of the journal element in the XSD specification of the return data, so that journal articles did not include the name of the periodical.

The in-use version of the Pico software has been thoroughly tested and is bug-free enough for ordinary use, while the version used for this experiment contains several improvements which still have to be integrated into the in-use application. Several tests had been previously run against the new features, but these were not sufficient to highlight those errors.

For completeness, we also ran the mutation testing against the instances generated by soapUI. Of course, these are not apt for testing, because they are completely independent of the aplication and populate the fields with very basic data, and without any variability (i.e., all generated instances for an operation are identical). Our experiments confirmed that such a test suite behaves very poorly, managing to kill only 23 mutants out of 153.

In conclusion, the experiments showed out that the systematic approach offered by WS-TAXI can provide, in a completely automated way, a test suite which is more effective than the one which is created manually. In particular, having a test suite which covers such a wide range of variability in the structure and the values of the data would require a huge effort if done manually, even starting from a basis of automatically generated skeletons such as those provided by soapUI. Even though we have not yet performed benchmark evaluation, the effort and time required appear drastically reduced using this methodology.

5 Related Work

In the introduction we have already cited some related industrial tools. We are now going to discuss the related work from a perspective more related to academic research. Notwithstanding the many open challenges in service testing, there are not as many papers as we could expect. Indeed, as noted in a 2005 report [16], *the area of testing services and their specifications has until now received limited attention from the research community.* Three years have passed from such claims, but in our opinion the situation has not changed a lot. We recently conducted a survey of several main conferences (i.e., ICSE, ICSOC, ICWS) and also workshops including "service-related" engineering activities within their lists of topics. We discovered that only around 30 papers on web service testing had been published until 2006. Making a cross-check, in its first edition, ICST, has a few papers on web applications and just one on services; symmetrically, the International Conference on Service-oriented Computing in its 2007 edition included only one paper on testing and the same was true for the 2006 edition. Therefore, we still need additional effort on WS testing research.

An interesting perspective on new challenges, opportunities and stakeholders in the testing of web services can be found in [7]. Among the various opportunities our approach tries to take advantage from the usage of standard descriptions and open formats to automatically manipulate and generate SOAP messages.

The direct manipulation of SOAP messages for testing purpose is also investigated by [12] which, to verify service behaviour, proposes to capture SOAP messages and to replay modified versions of the same message.

To the best of our knowledge, the only works which address issues similar to ours are [4] and [18]. The first work proposes XML-based test data generation and test operation generation. However, this work only outlines the possible perspectives of WSDL-based testing, but does not provide a tool, nor does it rely on standard test approaches. Similarly the second work permits to automatically generate SOAP messages from WSDL specification. Nevertheless this is done generating random values for the parameters in a invocation. Moreover the aprroach used does not seem easily exstensible to a document-style WS interaction paradigm. WS-TAXI offers a "turn-key" approach which focuses on a systematic generation of test cases based on the CP algorithm that is equally applicable to RPC or document-style interaction paradigm.

Finally, automatic generation of instances from XML Schemas is nowadays a feature of some, even commercial, tools [3, 21, 19]. Therefore in principle our approach could be pursued even using other XML instance generators then TAXI. Nevertheless we are not aware of any existing tool that tries to address the XML instance generation step in a systematic manner by applying well founded test strategies. Other interesting directions of investigation, related to the automatic derivation of instances, concern work done within the Model Driven Engineering community [8, 17]. However, since XML Schema does not contain constraints, the task of XML instance generation seems to be easier than model-based generation performed within MDE.

6 Conclusions and Future Work

Testing of Web Services is a challenging activity. Many characteristics (run-time discovery, multi-organization integration) of this new paradigm and its related technologies certainly contribute to make testing much more difficult. Nevertheless there are other characteristics that could be fruitfully exploited for testing purposes. Among these, the representation of data in a computer readable format (typically XML-based) facilitates the automatic derivation of data instances to be used for testing invocations.

Starting from this consideration we developed a methodology and a tool to automatically derive data instances from WSDL descriptions. Such data represent possible values that a real implementation of the service should be able to handle. The generated data instances are encapsulated in correct SOAP envelopes that can be used to invoke a service implementation. Furthermore we exploited the characteristics of an XML Schema-based data description to automatically apply well known testing methods such as Category Partition and boundary value selection. This results in the derivation of a suite of messages that are representative of the space of possible messages.

We experimented our approach on a real service implementation used to retrieve and store bibliographical data. The service was reasonably correct since already deployed and tested for some time. Therefore in order to check our approach we applied a mutant-based strategy using the existing service implementation as a trusted version (the oracle). As a baseline, we adopted a manually derived test suite; we then executed the two test suites, the one automatically derived by WS-TAXI and the one built by the expert tester, on the mutants. The results were encouraging. The automatically generated test suite killed about 70% of the derived mutants against about 43% killed by the manual one (being 65% more powerful). By increasing the number of WS-TAXI test messages, we were able to kill all non-equivalent mutants except two and, to our surprise, we were also able to identify two errors in the trusted version that had not been discovered before.

Main tasks for the future include: to perform more extensive evaluations of other case studies in order to identify fault categories that are easily discovered and better define the usage scope; to extend WS-TAXI functionality so as to also generate non-compliant test cases that can support robustness testing of the invoked service; to improve WS-TAXI implementation and make it available to the community for free download and experimentation.

References

[1] Automated combinatorial testing for software - beyond pairwise testing. `http://csrc.nist.gov/groups/SNS/acts/index.html#combinatorial`.

[2] Pico 5. `http://pico.sssup.it/`, 2006.

[3] Altova. XML Spy. `http://www.altova.com/products/xmlspy/xml_editor.html`.

[4] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen. WSDL-based automatic test case generation for web services testing. In *Proc. of IEEE Int. Work. SOSE*, pages 215–220, Washington, USA, 2005. IEEE Comp. Soc.

[5] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. Towards automated WSDL-based testing of web services. In *Proc. ICSOC 2008*, 2008. to appear.

[6] A. Bertolino, J. Gao, E. Marchetti, and A. Polini. Automatic test data generation for XML Schema based partition testing. In *Proc. Int. Work. on Automation of Software Test (ICSE'07 companion)*, Minneapolis, Minnesota, USA, May 2007.

[7] G. Canfora and M. Di Penta. Testing services and service-centric systems: challenges and opportunities. *IEEE IT Prof.*, 8(2):10–17, March/April 2006.

[8] K. Ehrig, J. Küster, G. Taentzer, and J. Winkelmann. Generating instance models from meta models. In *FMOODS'06 (Formal Methods for Open Object-Based Distributed Systems)*, pages 156 – 170, June 2006.

[9] eviware. soapUI; the Web Services Testing tool. `http://www.soapui.org/`. accessed 2008-05-30.

[10] Gartner. 2007 Press Release: Bad Technical Implementations and Lack of Governance Increase Risks of Failure in SOA Projects. `http://www.gartner.com/it/page.jsp?id=508397`. accessed 2008-09-30.

[11] Global Information Inc. SOA infrastructure market shares, market strategy, and market forecasts, 2008-2014. `http://www.the-infoshop.com/study/wg64381-soa-infra-mkt.html`, 2008. acc. 2008-05-22.

[12] J. Offutt and W. Xu. Generating test cases for web services using data perturbation. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, 2004.

[13] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686, 1988.

[14] Parasoft. SOATest. `http://www.parasoft.com/jsp/products/home.jsp?product=SOAP`. acc. 2008-06-03.

[15] PushToTest. PushToTest TestMaker. `http://www.pushtotest.com/Docs/downloads/features.html`. accessed 2008-06-03.

[16] A. Sassen and C. Macmillan. The service engineering area: An overview of its current state and a vision of its future. Technical report, EU Commis., July 2005.

[17] S. Sen, B. Baudry, and J.-M. Mottu. On combining mullti-formalism knowledge to select test models for model transformaion testing. In *IEEE 1st ICST*, Lillehammer, Norway, April 2008.

[18] H. M. Sneed and S. Huang. The design and use of WSDL-test: a tool for testing web services. *Journal of Software Maintenance and Evolution: Research and Practice*, 19:297–314, 2007.

[19] Sun. Sun XML Instance Generator. `http://wwws.sun.com/software/xml/developers/instancegenerator/index.html`, 2003.

[20] TAXI. Testing by Automatically generated Xml instances. `http://labse.isti.cnr.it/index.php?option=com_content&task=view&id=94&Itemid=49`.

[21] Toxgene. Toxgene. `http://www.cs.toronto.edu/tox/toxgene/`, 2005.

[22] WWW Consortium. WSDL version 2.0. `http://www.w3.org/TR/wsdl20/`, 2007.